LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-677974

Characterization of Proxy Application Performance on Advanced Architectures: UMT2013, MCB, AMG2013

L. H. Howell, B. T. Gunney, A. Bhatele

October 9, 2015

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Characterization of Proxy Application Performance on Advanced Architectures: UMT2013, MCB, AMG2013

Louis Howell, Brian Gunney, Abhinav Bhatele

Lawrence Livermore National Laboratory, Livermore, CA

Abstract

Three codes were tested at LLNL as part of a Tri-Lab effort to make detailed assessments of several proxy applications on various advanced architectures, with the eventual goal of extending these assessments to codes of programmatic interest running more realistic simulations. Teams from Sandia and Los Alamos tested proxy apps of their own. The focus in this report is on the LLNL codes UMT2013, MCB, and AMG2013. We present weak and strong MPI scaling results and studies of OpenMP efficiency on a large BG/Q system at LLNL, with comparison against similar tests on an Intel Sandy Bridge TLCC2 system. The hardware counters on BG/Q provide detailed information on many aspects of on-node performance, while information from the mpiP tool gives insight into the reasons for the differing scaling behavior on these two different architectures. Results from three more speculative tests are also included: one that exploits NVRAM as extended memory, one that studies performance under a power bound, and one that illustrates the effects of changing the torus network mapping on BG/Q.

Introduction

Modern computer architectures pose major challenges for simulation code development, not only requiring large-scale parallel scalability but also increasing attention to on-node parallel efficiency and memory management. Efforts are under way at many sites to better understand how codes of programmatic interest behave on advanced hardware both in order to make the implementations of key algorithms more efficient and to guide procurements of new machines. A team at LLNL has studied three proxy applications that include simplified implementations of important algorithms: UMT2013 for deterministic radiation transport on an unstructured mesh; MCB for Monte Carlo radiation transport; and AMG2013 for algebraic multigrid [1].

The LLNL effort was part of Tri-Lab ASC L2 Milestone #4875, "Evaluate Application Performance on Advanced Architectures". Presentations were given by teams from Sandia, LLNL, and LANL at Sandia in August 2014, after which the milestone was considered completed. Each lab was expected to report on two proxy applications. Though we did tests on three proxy apps, we

decided that the most conclusive results were for UMT and MCB and so focused on those for the milestone. This report expands on the LLNL experiments and conclusions and includes additional MCB tests and a section on AMG that were not part of the original presentation.

Most of the results presented here are for the IBM BG/Q architecture. The runs were done on Vulcan, part of the LLNL Sequoia procurement. The hardware counters on BG/Q provide detailed information on memory bandwidth, cache utilization, instruction mix, and other aspects of on-node performance [2]. For comparison, runs were also done on the Cab machine, a Linux cluster using 8-core Intel Xeon E5-2670 Sandy Bridge processors with InfiniBand QDR interconnect. Cab is part of the TLCC2 procurement. The mpiP [3] and memP [4] tools were used on both machines to give detailed information about MPI efficiency and heap memory use. One additional test used Catalyst, an Ivy Bridge based machine with unusually large memory (128GB per node) and an additional 800GB of SSD NVRAM per node. Software support for using the SSD memory without major code modifications has recently become available, but test results using the SSD memory show disappointing performance.

We will start with the basic MPI scaling, thread performance, and hardware counter results for UMT, then will move on to show related data for MCB and AMG. The tests for the three codes are similar but not identical, both because of differences among these three applications and because the different people running the tests explored different performance characteristics and expressed their results in somewhat different ways. (For the most part Howell ran the UMT tests, Gunney focused on MCB, and Bhatele did AMG, though there were some exceptions to this.) Some more unusual and speculative tests will appear after the main sections.

An additional aim of these tests was to better understand the performance tools themselves and issues encountered during the assessment process. None of the tools were perfect. We will mention some difficulties with the tools in the context of the individual tests, and will summarize some lessons learned in a short section at the end of the report.

UMT2013 Weak Scaling

A weak scaling test measures parallel performance in a series of runs where the size of the problem increases in proportion to the number of processors used, so the problem size per processor is held fixed. On Vulcan UMT2013 showed fairly good weak scaling to at least 131072 cores. At the largest scale the speed per iteration was 67% of that at small scale. (There was some additional slowdown due to the fact that the code required more iterations to converge at large scale. There were also issues with the scaling of the measurement tools themselves.) Results on Cab were similar but were only run out to 4096 cores. The parallel efficiency for the largest runs on Cab was comparable to that of the largest runs on Vulcan, though the latter used 32 times as many cores (and 128 times as many threads)!

In UMT2013 large-scale MPI parallelism is over 3D spatial domains. This discrete ordinates (S_n) code for multigroup deterministic transport [5, 6] also allows user control over the number of energy groups and the number of angles used in the discretization, and OpenMP threading is used over the angles. (The angles represent the directions in which photons are moving. In any spatial zone there can be radiation moving in many different directions with different intensities, and the set of angles can be considered a discretization of these directions over the 2D surface of the unit

sphere. Combined with the 1D spectrum of photon energies and the 3D spatial discretization the complete system is then 6 dimensional, plus one more for time. Since intensities must be stored for every combination of zone, angle, and energy, it is no surprise that memory management is a major consideration for this code.)

On Vulcan we did two series of weak scaling tests, one using $10 \times 10 \times 10$ and one using $12 \times 12 \times 12$ spatial grids per MPI task. Both problems used 16 energy groups and 256 angles. The runs were done with 8 MPI tasks per BG/Q node and 8 threads per task. Vulcan has 16 cores per node with 4 hardware threads per core, so this configuration made full use of the hardware threads. On Cab we ran the series with $12 \times 12 \times 12$ grids, using 16 MPI tasks per node and 1 thread per task. Cab has 16 cores per node and no hardware threads. The reason for running half as many MPI tasks per node on Vulcan was that this machine has only 16GB of memory on each node, compared to 32GB for Cab; running 16 tasks per node on Vulcan would require smaller grids per task. The runs on Vulcan used roughly 7.5GB per node for $10 \times 10 \times 10$ grids and 12.5GB for $12 \times 12 \times 12$, as measured by memP. The Cab runs used 26.4GB per node. (Comparisons exploring the performance effects of OpenMP threading in detail appear in later sections below.)

Runs were instrumented using the mpiP tool [3] for MPI profiling and the memP tool [4] to measure heap memory use. Figure 1 shows the performance results in three different ways. First we look at the figure of merit, a metric that focuses on the rate at which basic components are executed and factors out the numbers of iterations required for convergence. (The iteration count is also an important component of overall runtime, but our main concern here is with the performance of this *implementation* on particular architectures, as opposed to the mathematical properties of the algorithm.) Perfect weak scaling would yield flat lines. The Vulcan (BG/Q) results are not perfect but are still fairly good. The Cab (TLCC2) results show a steeper drop in performance at scale, even on the much smaller number of available processors, though individual processors are faster than those on BG/Q. The difference between the $10 \times 10 \times 10$ and $12 \times 12 \times 12$ tests is negligible.

The second plot (bottom left) shows the percentage of time spent in MPI communication as measured by mpiP. Various mpiP options were tried. The `-c` option combines results from all MPI tasks into a single report, `-l` switches to a different algorithm for combining information from different tasks, and `-o` turns off profiling from the beginning of execution, instead relying on explicit calls we put in to activate the instrumentation only during the main timestep loop. The `-o` test did not work on Cab because the activation calls failed to turn on the MPI profiling. (On the TLCC2 machines UMT2013 builds using dynamic libraries, and apparently picks up the wrong version of the activation routines when linking with mpiP.)

The MPI profiles on Vulcan show a small and roughly constant percentage of time going to MPI communication, independent of grid size and the mpiP options. Cab, on the other hand, shows an increasing amount of time going to MPI as the number of tasks grows. Apparent differences between the two curves generated for Cab are not repeatable and therefore not really caused by the different mpiP options tested for this plot. All timing results on Cab vary more from one run to the next than those measured on Vulcan. We will demonstrate this in more detail in the strong scaling section below.

The third figure (bottom right) looks closely at the costs of running the instrumentation itself. All runs are for the $10 \times 10 \times 10$ series of tests on Vulcan, with separate curves for the code without instrumentation, with the various mpiP options, with the memP memory profiling tool, and with

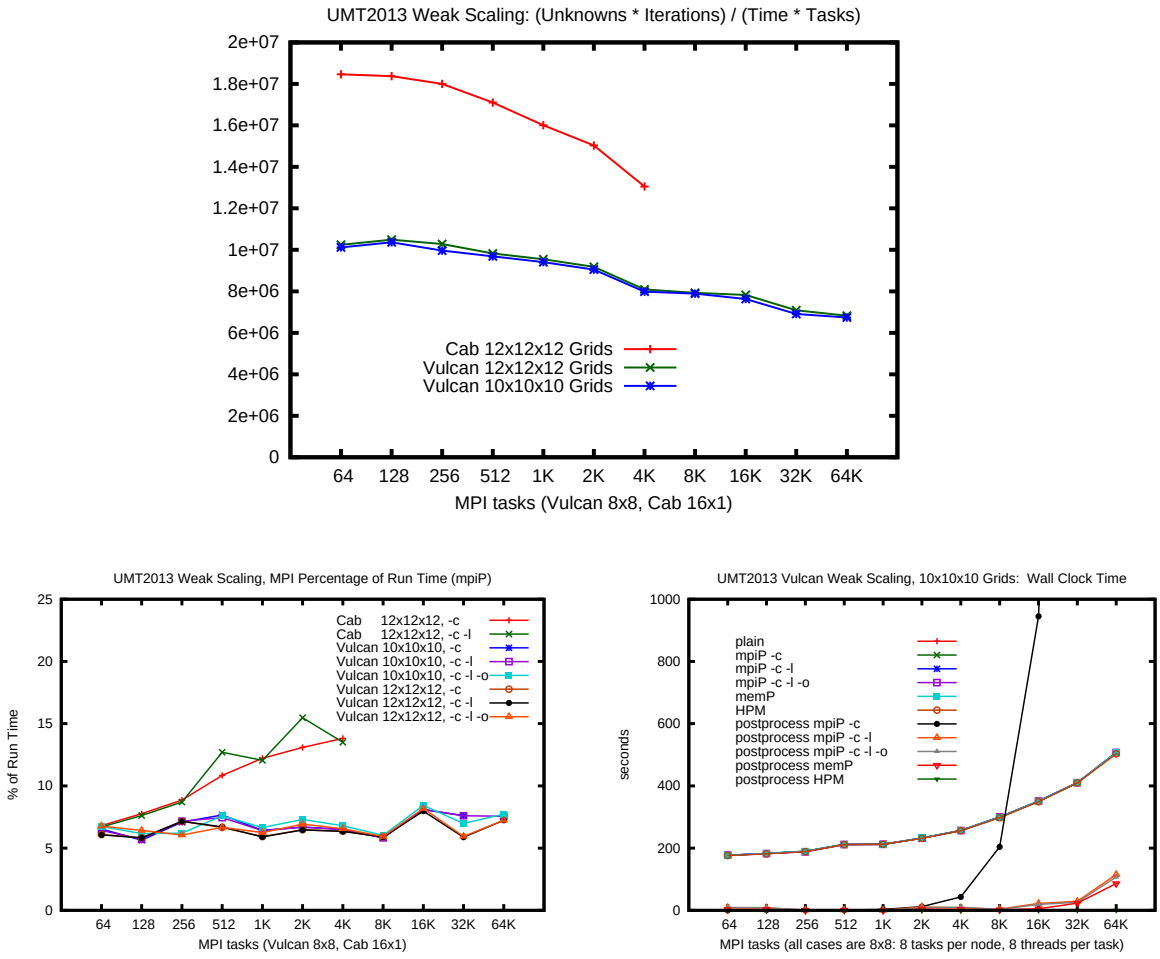

Figure 1: Weak scaling study. Top: Figure of merit. Bottom Left: Percentage of run time tied up in MPI communication. Bottom Right: Wall clock times showing overheads caused by the performance instrumentation.

the HPM (mpitrace) library for gathering information from the BG/Q hardware performance counters. (We will show results from the HPM library in a later section.) These are wall clock measurements, and a major component of the upward trend for the six main curves is the increase in the number of iterations required for convergence at larger problem sizes. All six curves for the main code lie exactly on top of one another, which shows that the tools are not significantly perturbing these timing measurements. Where the tools do show a significant cost, though, is in the post-processing phase after the main part of UMT has finished. Post-processing costs for mpiP skyrocket without the `-l` option. (Memory costs for this phase also rise dramatically. Tests for the series with $12\times12\times12$ grids were closer to filling memory and often crashed without `-l`.) The post-processing costs for mpiP with `-l` are smaller but are also starting to rise for the largest runs; the same is true for memP. The overheads added by HPM library were exceptionally low compared to those from the other tools, both during the collection phase and during postprocessing.

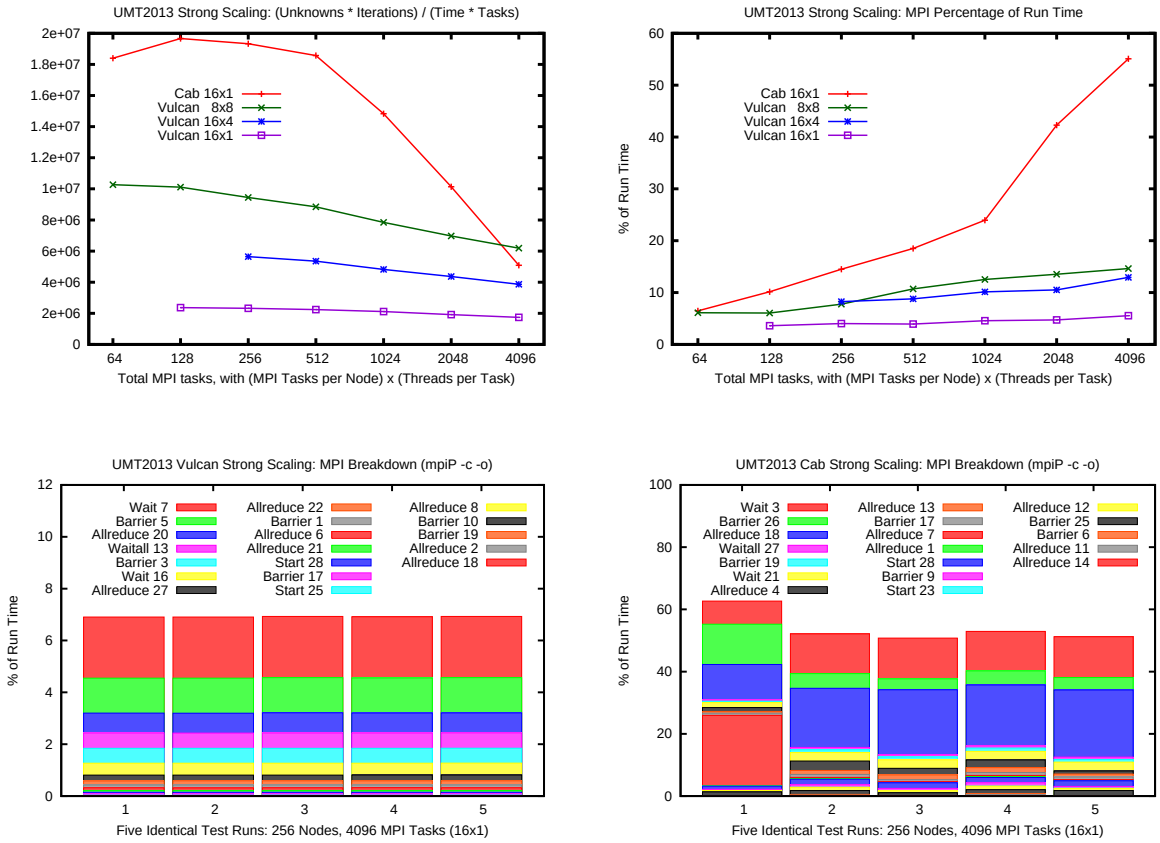

Figure 2: Strong scaling runs on Vulcan (BG/Q) and Cab (TLCC2), with the Vulcan results also demonstrating reasonably efficient use of OpenMP threads. Top Left: Figure of merit. Top Right: Percentage of run time tied up in MPI communication. Bottom: Comparison showing five identical runs of the largest 16 x 1 problem on each machine, with MPI time during the timestep loop broken down by callsite. The callsites are arranged in the same order for both runs even though mpiP identifies them with different labels. Note the different scale and greater variability for the tests on Cab. Each series of five tests was run with a single script and therefore used the same processors under similar runtime conditions.

UMT2013 Strong Scaling

A strong scaling test measures parallel performance with the total size of the problem held constant. In UMT2013 MPI parallelism is over 3D spatial domains, so as the number of processors increased we reduced the spatial grid size on each processor to keep the total number of unknowns unchanged. The tests shown in Fig. 2 used grids varying from $12\times12\times12$ down to $3\times3\times3$. Users also have control over the number of angles (transport directions) and energy groups, but for this test we held these constant. Note that OpenMP threading is over the angles in each octant and so is independent of the spatial grid size.

Both Vulcan and Cab have 16 cores per node, but on Vulcan there are also 4 hardware threads per

core. OpenMP threading allows access to these hardware threads and can also be used for running multiple cores per MPI task. All of the Cab runs shown here used 16 MPI tasks per node and 1 thread per task (16×1), but the Vulcan runs were done in three groups: a (16×1) series was similar to the Cab runs but did not use all the hardware threads, a (16×4) series was faster because it did use all the hardware threads, and an (8×8) series was faster still because it used twice as many cores as the other two. The large spread between the different Vulcan curves is not a surprise since these tests used different machine resources: different numbers of cores and hardware threads. For a comparison showing the effects of allocating the same resources in different ways see the next section.

The top left plot in Fig. 2 shows the figure of merit for the central timestep loop: with perfect strong scaling the lines would be flat. Cab is faster for small runs but shows much worse parallel scaling than Vulcan. The top right plot shows how much time in each run was spent doing MPI communication, as reported by mpiP. (These mpiP measurements are for the whole code, not just the central timestep loop, but that doesn't make a major difference here.) The obvious conclusion is that Vulcan has slower processors but a better interconnect. The bottom two plots, though, show that there is another factor involved that may be even more critical at scale. Times spent in each MPI communication step are much more variable on Cab than on Vulcan from one run to the next. Operating system noise is a likely explanation: when interrupts occur during an MPI collective the processors that are delayed keep all the others waiting. Contention for network resources with other jobs running on the machine is another contributing factor. The BG/Q network is better at isolating jobs from each other.

These MPI breakdown plots use the $-o$ option to mpiP in order to filter out some variability that happens during initialization and focus on the main timestep loop. To get $-o$ to work on Cab we changed the build to use static libraries. Though mpiP uses different numbers to identify MPI callsites on the two different machines, we have arranged the figures so that corresponding callsites appear in the same order and can be compared directly.

UMT2013 Threading and Hardware Counter Results

The examples in the previous section showed that OpenMP threading improved performance on Vulcan for a fixed number of MPI tasks by using additional cores and hardware threads per task. It is a more complicated question whether MPI or OpenMP threads provide the more efficient use of a fixed allocation of hardware resources. This section explores the tradeoff between MPI and OpenMP parallelism, and also highlights some of the detailed information available from BG/Q hardware performance counters that helps interpret the differences between tests.

In the top row of Fig. 3 we show the performance for 64 nodes of Vulcan and of Cab as the tradeoff varies from all-MPI to all-OpenMP on each node. Though there are 16 cores per node on each machine, the 64 hardware threads on each BG/Q node can be accessed in any MPI/OpenMP combination from (64×1) to (1×64). On Cab the combinations vary from (16×1) to (1×16). The figure of merit used for these plots is slightly different from the one used in the scaling studies: since the amount of machine hardware is held constant we do not divide by the number of MPI tasks. Also note that because each test keeps the total number of mesh zones constant, but varies the number of MPI tasks, the size of the spatial domain handled by each MPI task is changing as we move from left to right across each plot.

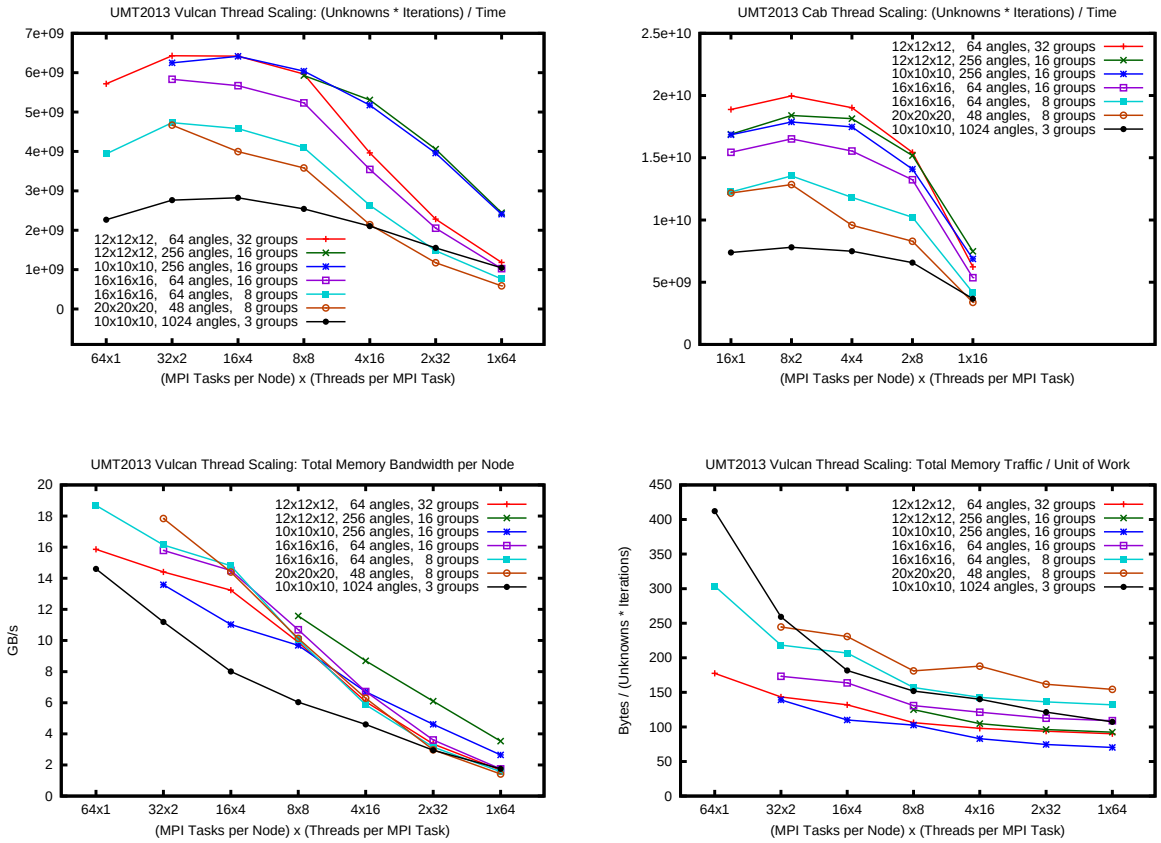

Figure 3: Performance for a fixed number of processors, showing changes as the code shifts from pure MPI parallelism to heavy use of OpenMP threading. Top: Figures of merit on Vulcan and on Cab. Bottom Left: Memory bandwidth per node on Vulcan. Bottom Right: Memory traffic per work unit.

Several runs are shown with variations in spatial grid size and numbers of angles and energy groups. (The grid sizes shown in each legend are the ones corresponding to 8 MPI tasks per node. So on Vulcan a line labeled "$12\times12\times12$" actually uses grids that vary from $3\times3\times3$ to $24\times24\times24$ per MPI task as the number of tasks changes.) High figures of merit in these plots tend to result from using large numbers of energy groups. All energy groups share the same geometric quantities associated with a given combination of zone and angle, so in UMT2013 the loop over energy groups is the innermost loop. With more groups the geometric calculations are amortized over more unknowns. The lowest curve is the one using only three energy groups. This is also the flattest curve, though. It runs more efficiently with large numbers of OpenMP threads because it has a large number of angles to thread over. The other curves show more of a drop in performance for large numbers of threads because they don't have enough angles per octant to keep all the threads busy.

The results for Cab show a sharper drop in performance for the (1×16) mode, even in the cases with large numbers of angles that perform well with up to 64 threads on Vulcan. There are two sockets on each node of Cab. Threading works well across the 8 cores on each Xeon chip, but

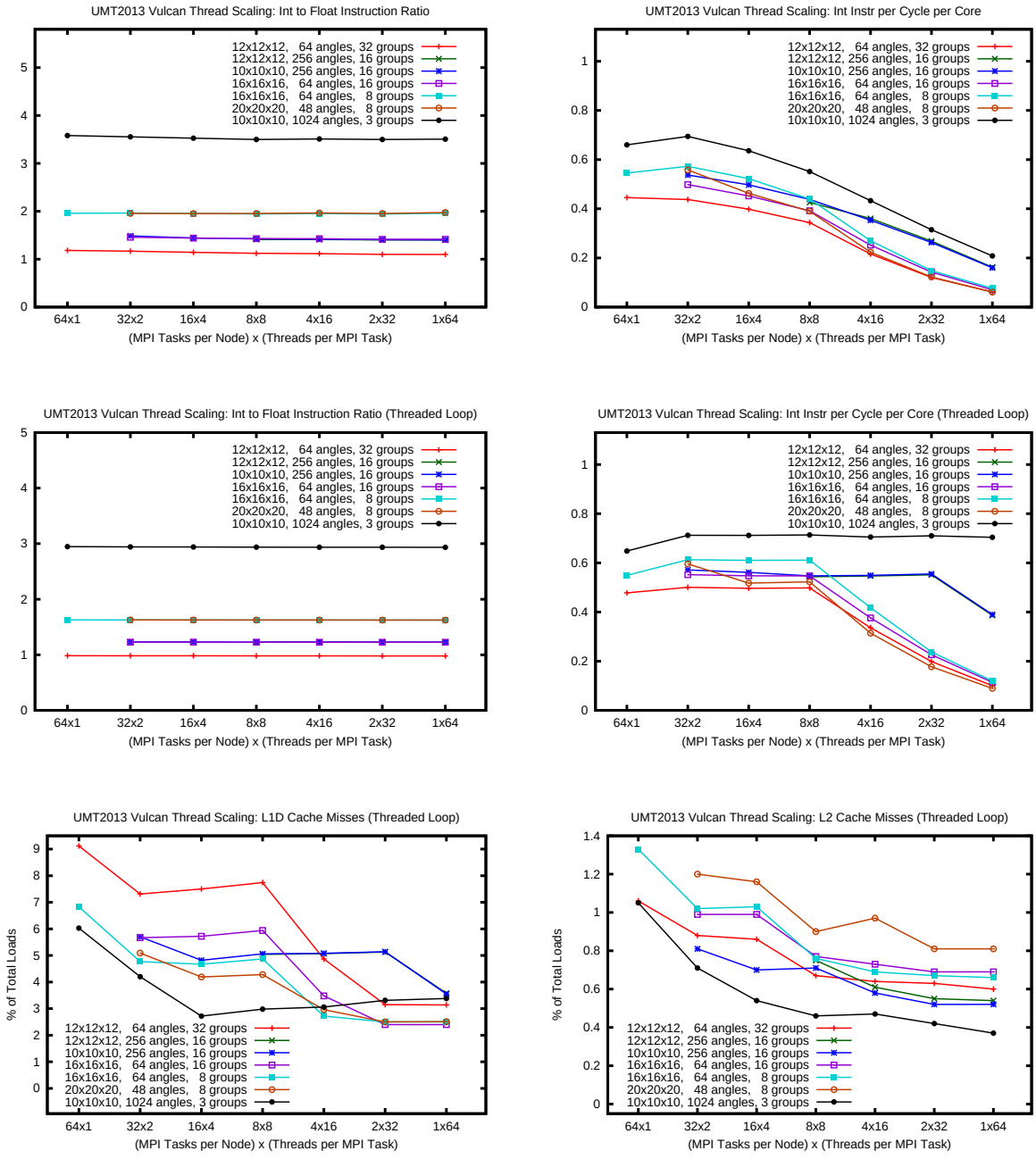

Figure 4: BG/Q hardware performance counter information showing effects of shifting from pure MPI to OpenMP for a fixed number of processors. Top: ratio between int and float instructions, and int instructions per cycle, measured over the full timestep loop which includes some unthreaded code. Middle: same quantities measured only for the threaded loop over angles. Bottom: L1 and L2 cache miss rates for the threaded angle loop, which help explain why some test cases achieved higher instruction rates than others.

threading across the entire node is much slower due to low bandwidth between the two sockets.

The second row of Fig. 3 shows the use of memory bandwidth for UMT2013 on BG/Q, as reported by the HPM (mpitrace) library [2]. The total bandwidth available on a BG/Q node in practice is about 28 GB/s. A previous version of UMT was bandwidth-limited, but this was changed by a revision in the data structure layout in UMT2013. None of the test cases here uses more than 19 GB/s. There is a strong and somewhat puzzling dependence on the number of threads per MPI task, with pure-MPI cases using the most bandwidth. To attempt to explain why bandwidth use declines so consistently with increasing thread count, the plot at lower right looks at memory bandwidth divided by the figure of merit (with a constant scaling factor). The result is the number of bytes of memory traffic required to execute each iteration of the algorithm, per unknown. The downward trend in these curves for small numbers of threads can be interpreted as different threads sharing data fetched into cache. For larger numbers of threads, though, the curves flatten out, showing that the continued decrease in memory bandwidth is tied to the decrease in the figure of merit: the code draws less memory simply because it is doing less useful work. The apparently steady decline in memory bandwidth is thus seen to be a combination of two different effects for different parts of the curves.

As a separate issue, the memory footprint was also higher for cases with more MPI tasks, even though all tests on each curve have the same number of unknowns. Missing points on some of the curves were runs that failed due to insufficient memory. All of these missing points are on the left sides of the various plots. We're not including plots of memory usage because the results are somewhat irregular. The memP tool does not appear to be thread-safe, and becomes more and more likely to return corrupt results as the number of threads increases. There is also a heap memory estimate returned by mpitrace, but this is consistently higher than the memP estimate even in cases where memP appears to be functioning correctly. It appears that that mpitrace reports the heap memory allocated to the process by the operating system, while memP reports the amount actually used by the application.

The "overloaded" modes (64×1) and (32×2) on BG/Q use multiple MPI tasks per core. It is surprising that these perform even as well as they do. The heavy use of both memory and memory bandwidth in these modes, though, indicates that they do not make effective use of machine resources and will not be practical for most production calculations.

The six plots in Fig. 4 show more information about these same runs obtained using the BG/Q hardware performance counters. The top two plots were derived from measurements over the full timestep loop, which includes both threaded and unthreaded code. The ratio between integer and floating point instructions does not depend at all on the number of threads per task but only on the number of energy groups. Integer instructions here are mostly indexing operations: striding through the mesh. All of the test cases use at least as many int as float instructions. The highest floating point usage occurs with large numbers of energy groups because these cases have more work per mesh zone (recall that the loop over groups is the innermost loop). A BG/Q core can perform at most one int and one float instruction per cycle, and it is unusual to see rates much above 0.7 in practice. The plot on the right shows that the int instruction rate can reach a large fraction of the maximum rate when the number of threads is small. The integer instruction rate appears to be a primary limiting factor for UMT2013.

The second row of plots shows the same quantities measured only for the threaded loop over

angles. The integer instruction rate is instructive. Each downturn in each of the curves happens when the number of threads exceeds the number of angles per octant. Without this effect the curves would be essentially flat. The plot above this one, that measures the same quantity over the full timestep loop, shows a more steady decline in the instruction rate. This decline is therefore associated with the unthreaded code outside the angle loop, which can dominate the run time when the number of threads is large.

The final two plots in Fig. 4 give L1 and L2 cache miss rates as a fraction of loads. These help explain the remaining differences between integer instruction rates for the various test cases. Some features of the instruction rate curves are due to idle threads, as discussed above. The remaining differences correlate with cache miss rates, and tests with higher instruction rates generally showed lower cache miss rates. The lowest cache miss rates were for the test problem with the fewest energy groups. This is the case that did the most integer operations per floating point operation, reached the highest integer instruction rates, and drew the least bandwidth from main memory.

There are many other counters available in addition to the ones used for the figures. Almost no SIMD instructions are generated by UMT2013, and none at all in the threaded loop over angles where most of the work is done. If the code were modified to make better use of the SIMD units—perhaps through explicit directives—then the SIMD counter information would give valuable information about the effects of the change. The instruction cache miss rate turns out to be an issue for one of the other apps (MCB). UMT2013 has a very low rate of instruction cache misses, though, so this is not a performance issue worth exploring here.

MCB Weak Scaling

MCB is a 2D radiation transport code based on a Monte Carlo algorithm. Discrete particles are tracked as they move through the mesh, interacting with the continuum material through scattering and absorption. MPI parallelism is over spatial domains, threading is over local particles in each domain. Because the mesh and the test problem are simple and our tests ran on power-of-two core counts, we expected task-based parallel partitioning to be well balanced and to have high data locality. The excellent MPI weak scaling results confirm this assumption.

Each absorbed particle changes the state of the material by depositing energy, so the particle loop would not be thread-safe if threads could simultaneously write to the same material data location. MCB deals with this problem by replicating the local mesh domain for each thread. Each thread modifies its own copy. The separate thread copies of the deposition array are accumulated into a master copy by a single thread once particle tracking is completed. This threading model works well for small numbers of threads, but the synchronization step does not scale well as the number of threads per MPI task becomes large.

Our weak scaling benchmark had 200000 particles per core and 10000 zones per core with about 3.2 scatters per zone crossing. The scattering opacity was 20 times the absorption opacity, so a typical particle would scatter 20 times before being absorbed. We ran several different series of tests. In the milestone presentation we showed only figure-of-merit results for two machines, and only for modes from (16×4) to (1×64) on Vulcan and from (16×1) to (1×16) on Cab. Those plots are the top row of Fig. 5. As a later followup we ran the code with mpiP on both machines, and added the "overloaded" thread modes (64×1) and (32×2) on Vulcan. Results for these additional

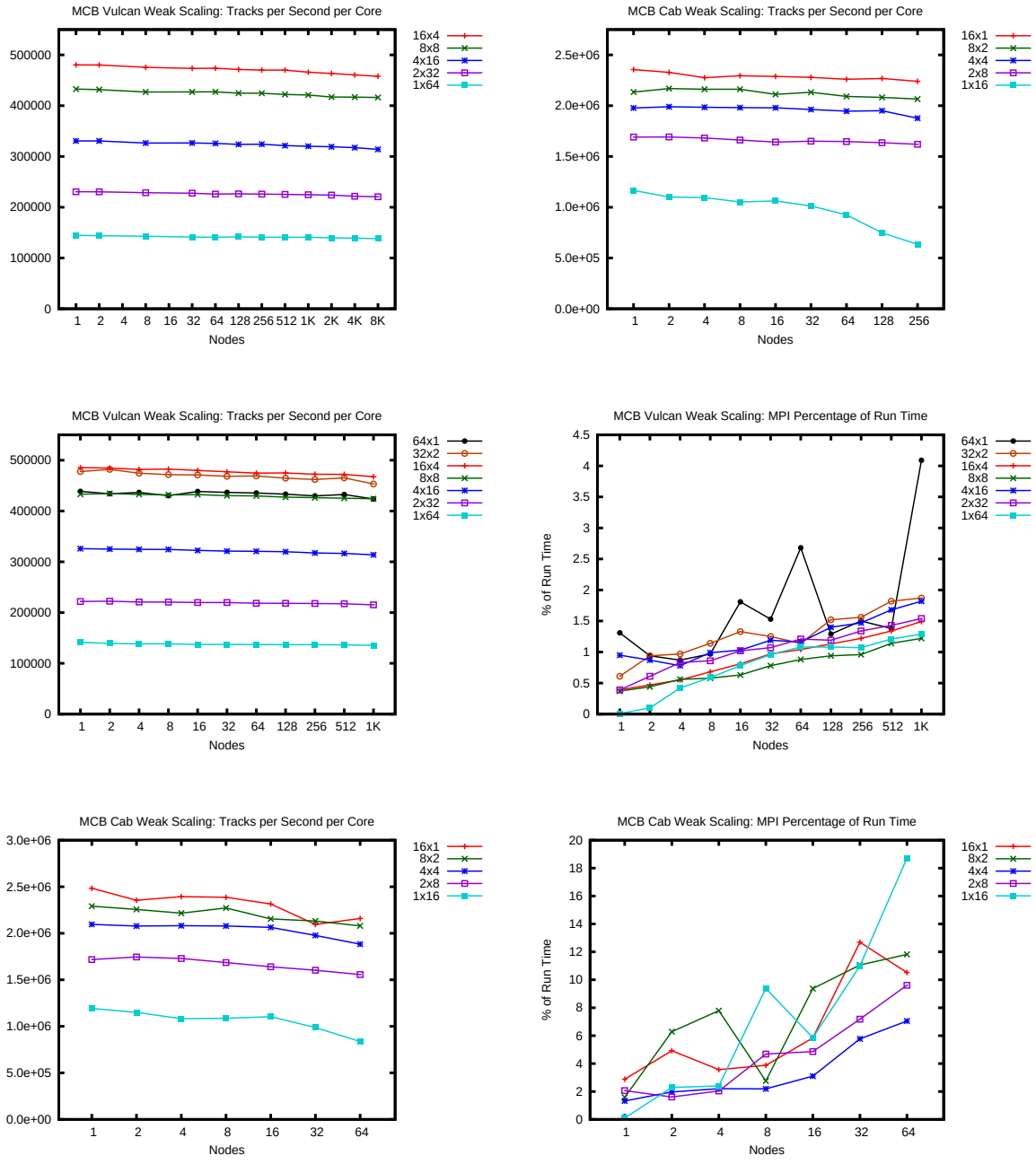

Figure 5: MCB weak scaling results. Top: Figure of merit plots for Vulcan and Cab from original presentation. Middle: Additional results for Vulcan showing overloaded modes 64 x 1 and 32 x 2 and the percentage of run time spent in MPI. Bottom: Additional results for Cab showing MPI percentage of run time. Note that both rows of additional results were run on fewer nodes than the original tests.

runs are the second and third rows of the figure. These followup runs were done on fewer nodes than the original tests, so we show both sets even though they contain some redundant information.

The figure of merit for MCB is tracks per second per core, where a track is the segment between events for a single particle. Events include zone crossings as well as any interaction with the background material. Vulcan had a peak performance of 480000 tracks/second/core, achieved on a single node using all 16 MPI tasks with 4 threads per task. Other thread modes were slower but only modes with 16 or more threads per task slowed down significantly. In all cases the performance stayed nearly flat as core count increased. Even the largest runs on Vulcan showed 95% parallel efficiency on 8192 BG/Q nodes (131072 cores), independent of thread mode. The mpiP tests show only a modest increase in the time spent in MPI communication up to 1024 nodes. (The (64×1) mode, using 4 MPI tasks per core, is a minor exception. MPI timings were more erratic in this mode for reasons we have not explained, though not enough to seriously impact the overall performance results.)

In contrast, Cab had a peak figure of merit roughly 5 times that of Vulcan, but with somewhat poorer MPI performance. Most thread modes showed good weak scaling out to 256 nodes. The (1×16) mode was significantly slower—as we would expect based on the UMT performance shown in Fig. 3—and also scaled more poorly. The MPI percentages of run time were much higher and more erratic than on Vulcan for all thread modes, even for small numbers of MPI tasks.

MCB Strong Scaling

For strong scaling we ran a benchmark with a constant global size of 51200000 particles in a 768×768 mesh. It had an average of 86.8 particles per zone and 6.667 scatters per zone crossing. Run on a range from 1 to 1024 Vulcan nodes, this translates to a high of 36864 zones and 3200000 particles per core down to a low of 36 zones and 3125 particles per core. On Cab we only ran out to 256 nodes, giving a low of 144 zones and 12500 particles per core.

On the BG/Q machine strong scaling was good out to 256 nodes for all thread modes, and even past this point the drop in performance was not precipitous. The relative efficiency of the different thread modes was similar to what we saw in the weak scaling test: the (16×4) mode was always the fastest and the modes with 16 or more threads per task were always slowest. The MPI percentage of run time tended to be higher for the tests with more MPI tasks, which is not surprising but confirms that the relative performance of different thread modes is unrelated to MPI communication time.

Results from Cab were more erratic and showed a much higher percentage of time spent in MPI, and the poorest performance and scaling behavior were for the (1×16) mode, all as we would expect based on the behavior of other tests. One surprise, though, is that in the strong scaling limit the performance of the pure-MPI (16×1) mode actually improved, and hints of similar behavior are visible in the plots for some but not all of the other modes. Our best explanation is that the extremely small problem sizes per core in the strong scaling limit may have fit better into cache.

(As with weak scaling, the original runs shown in the milestone presentation lacked the (64×1) and (32×2) thread modes on Vulcan. We later repeated the tests and included those modes. In this case the original plots show no additional information and so only the newer runs are shown here.

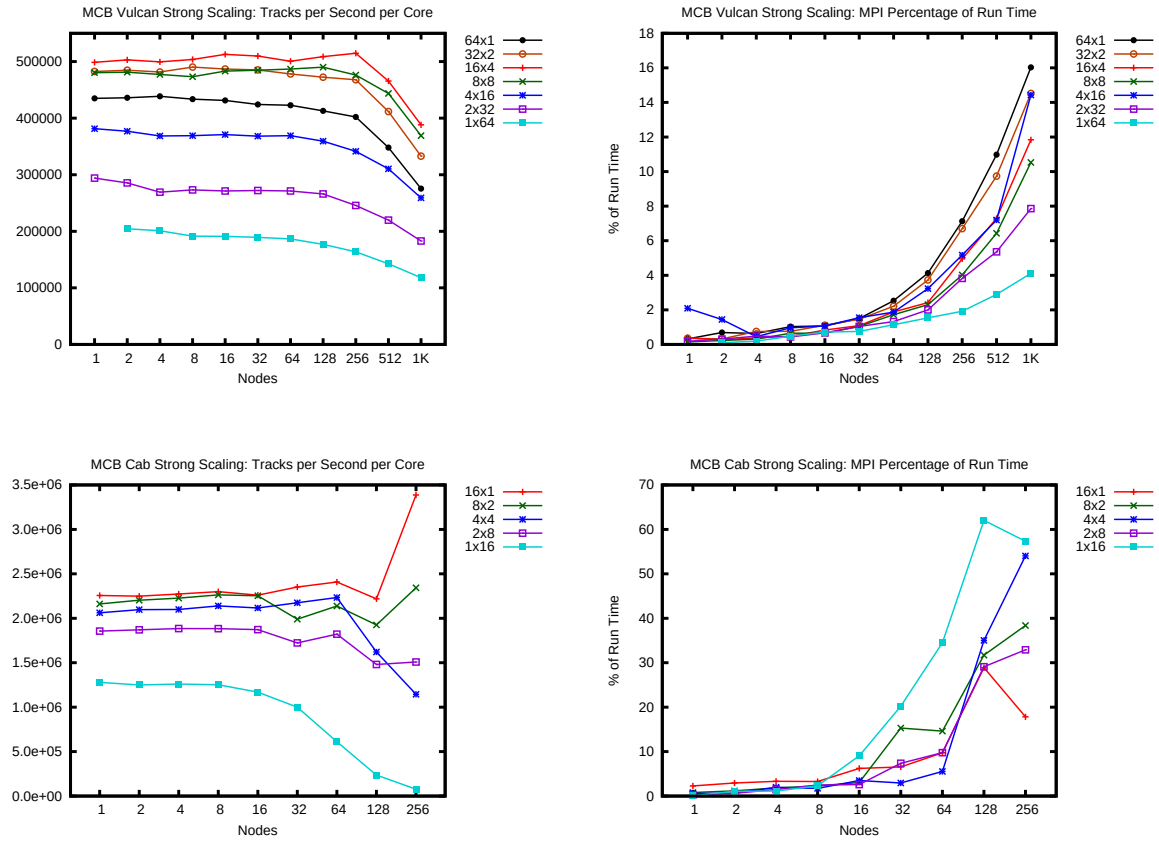

Figure 6: MCB strong scaling results. Top: Vulcan. Bottom: Cab. All runs on both machines used the same problem size, though on Vulcan this problem could be run on more nodes and with more threads per node. The missing data point on the Vulcan plots was a case that ran out of memory.

The minor uptick in MPI percentage visible for the (4×16) mode for 1 or 2 nodes did not appear in the earlier plots and so does not appear to be a repeatable measurement. Other minor glitches in the original tests disappeared in the newer runs. Vulcan results are more repeatable than those on Cab but not perfectly so.)

The percentage of time spent in MPI increased roughly as the square root of the core count, for both machines. This is not an unusual result. It correlates with the change in surface-to-volume ratio for the individual spatial domains, and thus with the expected ratio of message traffic to local computation.

MCB Single-Node Tests with Hardware Counter Measurements

Mesh resolution and particle count are independent parameters that determine the computational size of the problem, and MCB's performance depends on these parameters in different ways. We explored these effects by running the four combinations of 50000 or 200000 particles per core with

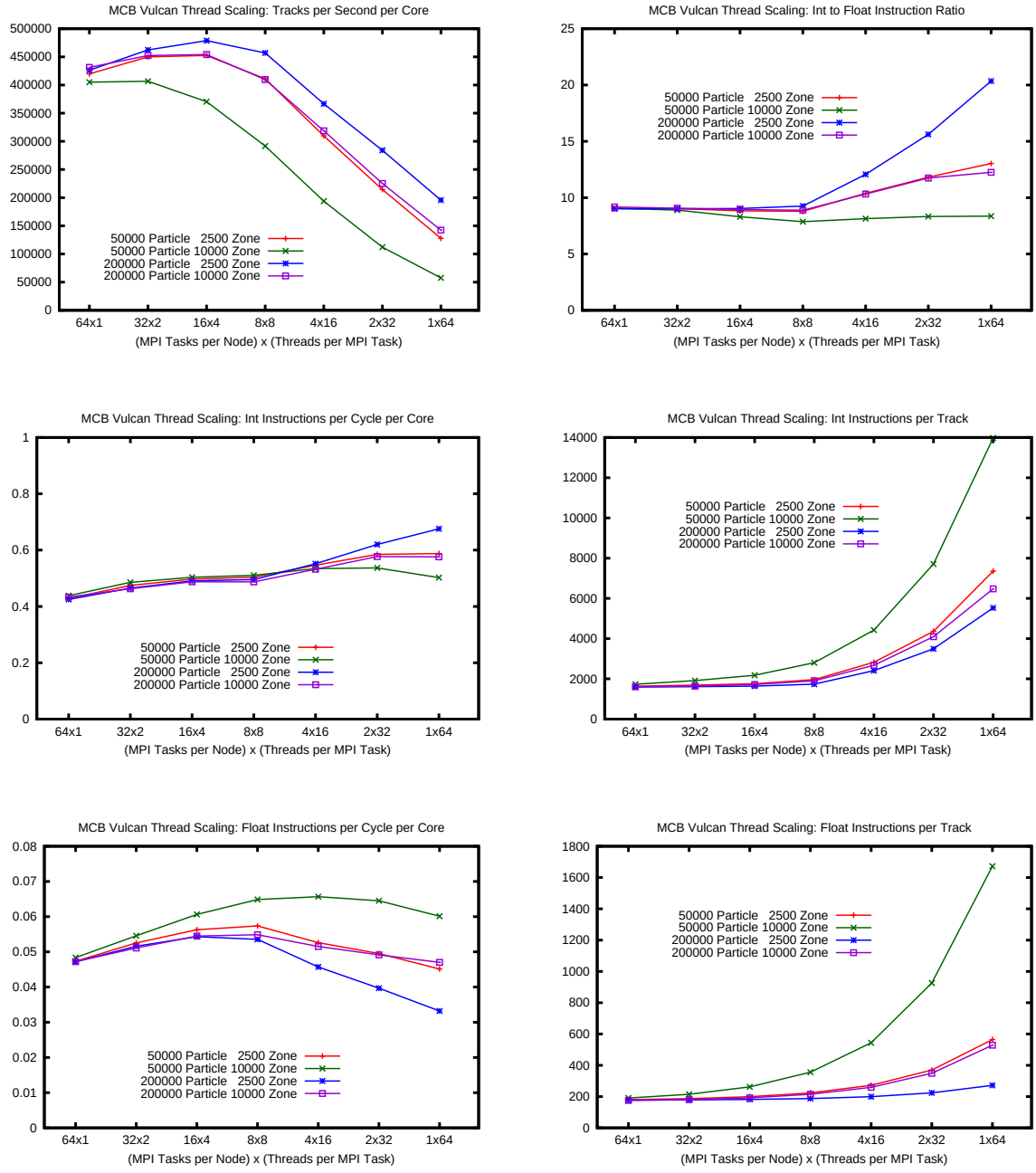

Figure 7: MCB performance on a single node of BG/Q, showing changes as the code shifts from pure-MPI to pure-OpenMP threading. The ratio of integer to floating point instructions is much higher than for UMT. Though the rate of computing tracks falls off with threading, the int instruction rate does not, the reason being that more instructions are being spent on combining the results from different threads. Floating point instructions per track also increase with threading but are driven more by the number of zones than the number of particles. Curve labels refer to the numbers of particles and zones *per core.*

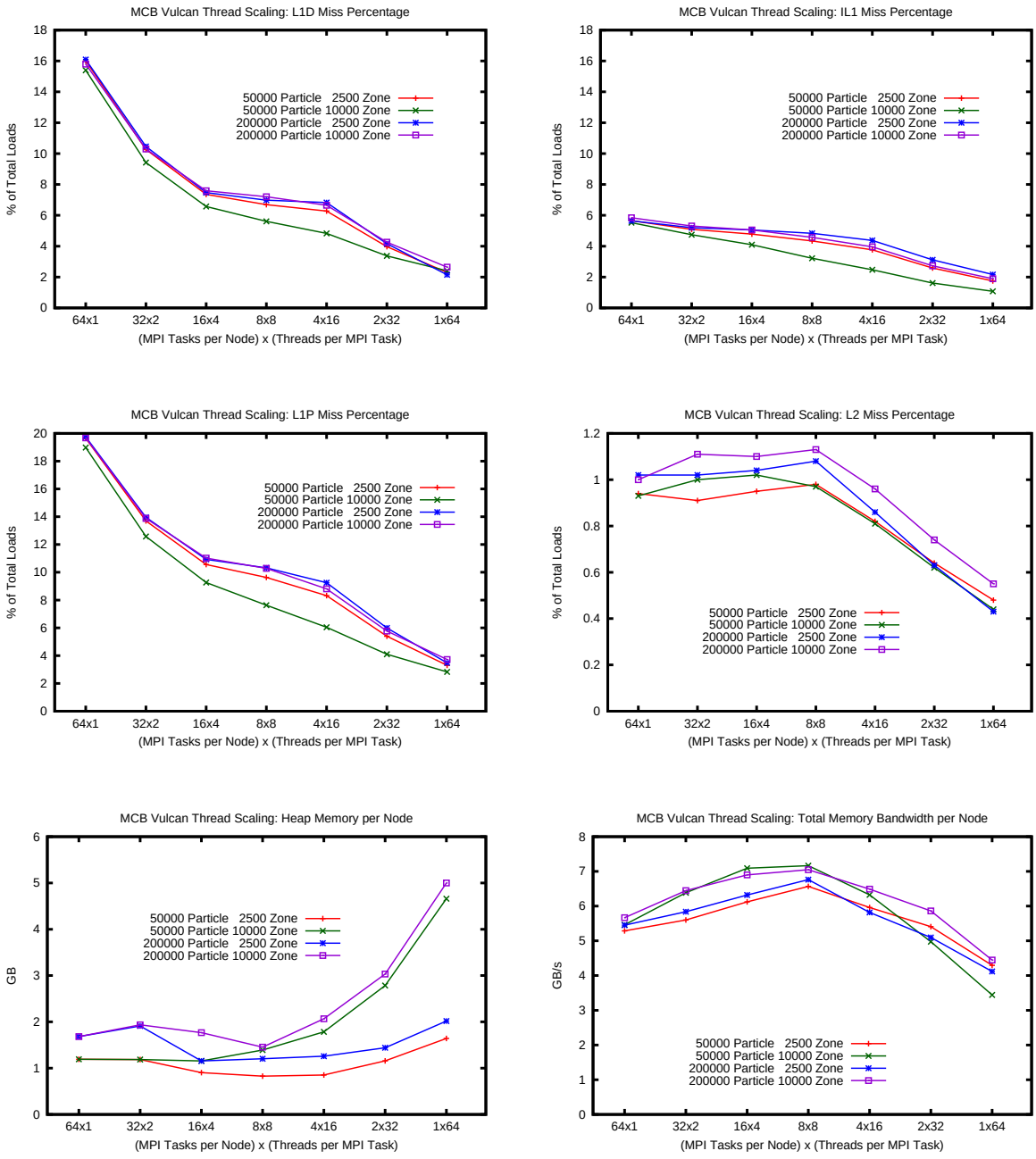

Figure 8: More MCB single-node results for BG/Q: L1 miss rates are high for both data and instruction caches. The L1P rate is also high, meaning that most loads that miss L1D or IL1 have to go to L2. The L2 miss rate is moderate, and as a result the use of main memory bandwidth averages about a quarter of the practical maximum of about 28 GB/s. Finally, the plot at lower left shows how the heap memory usage rises with the number of OpenMP threads, and how this increase is driven much more by the size of the mesh than by the number of particles in the problem.

2500 or 10000 zones per core. All combinations simulate the same physics—same opacity and same physical domain size—but not with the same accuracy. The high mesh resolutions corresponded to an average of 3.2 scatters per zone crossing, the lower to 6.4, because the relative number of zone crossings increases as zones become smaller. We ran each combination on a single Vulcan node, using all 64 available hardware threads but in different combinations of MPI tasks and OpenMP threads.

Examination of BG/Q hardware counter data helps to explain the changes in MCB performance for these different sets of parameters, just as it did for UMT. In Fig. 7 and 8 we show some of the relevant results. In the first plot we see that performance is best when the number of OpenMP threads is small, as we also saw in the scaling studies. This is similar to the behavior we saw for UMT. Unlike UMT, though, the numbers of instructions executed per cycle do not decrease with increasing thread count. For UMT performance dropped off because there were not enough angles to keep all threads busy at once. With MCB, as the number of threads per task increases the code may actually execute more instructions in a given time, but more of that time goes into mesh synchronization work that does not contribute directly to the figure of merit.

The highest performance curve is for the largest number of particles and the smallest number of zones per core. This is not surprising given that the figure of merit is based on the number of particle tracks computed, not on processing the mesh data. It says nothing about the best way to run the code to achieve a desired solution accuracy. In this study we're trying to understand code behavior at the machine level, not the algorithm level. Since we're not looking at solution accuracy there isn't enough information here to pick an optimal ratio of particles to zones. What we do gain from these comparisons, though, is additional insight into what parts of the algorithm are incurring which sorts of expenses.

The ratio between integer and floating point instructions is very high for MCB. The random numbers that govern all particle events are generated through an integer process. In addition, MCB computes energy deposition using integer arithmetic so that results will be repeatable for debugging purposes. (In floating point arithmetic the results would depend on the order in which different particles deposit their energy and so repeating a parallel calculation exactly would not be possible.) The dependence of the int to float ratio on the number of OpenMP threads reveals a curious pattern: At low thread counts all cases run at the same ratio, but for higher thread counts the instruction ratio depends on the numbers of particles and zones per core. Increasing the particle count drives up the relative number of int instructions, but increasing the zone count keeps the ratio more nearly balanced.

The integer instruction rate is a key limiting factor for MCB. Vulcan can execute at most one integer instruction (and one floating point instruction) per cycle per core. Rates seen in practice tend not to be much over 0.7 even for very efficient codes. The rates shown for MCB in Fig. 7 range from 0.4 to 0.65. These are not too far below the practical maximum. Cache miss rates, particularly the L1 caches, offer a plausible explanation for why they fall short—more on that below.

There is not a strong correlation between performance in terms of particle tracks and in terms of instruction rates. Fig. 7 shows rates for both integer and floating point instructions. The integer rate actually increases with increasing thread count, particularly the case with many particles and few zones. The floating point rate is much lower in absolute terms but also rises initially with

threading, as does the figure of merit in tracks per second, so at low thread counts the performance and instruction rates do appear to be coupled. Above 8 threads per task, though, the floating point instruction rate starts dropping again, and the case with many particles and few zones shows the steepest decline.

Plotting the instructions executed per track makes the differences stand out even more strongly. Integer and floating point work per track both increase for high thread counts, and the increases are both most pronounced when there are many zones and few particles per core. The apparently steady int to float ratio for this case in the upper left plot masks the fact that both integer and floating point instructions per track are *both* rising steeply. For the other cases the integer work rises faster than floating point.

It is clear from these results that there are code features that depend on the number of particles and others that depend on the number of zones, that these (possibly) separate code sections drive the hardware in measurably different ways, and that both are threaded in ways that scale well only to small numbers of threads. It is likely that at least part of the zone-dependent work is involved in combining the energy deposition arrays for the different threads, since these arrays increase in size with both the zone count and the number of threads. We have not attempted to instrument the separate sections of code in MCB; this would have to be done with care because it would require instrumentation calls within a threaded loop. Such tests, along with code modifications to try to improve thread performance, would be an obvious next step to take.

Figure 8 shows additional counter results for the same test problems. The L1D (data) cache miss rates are quite high for small numbers of threads. These "overloaded" modes with multiple MPI tasks per core also caused the highest miss rates for UMT, but not to the extreme we see here with MCB. The IL1 (instruction) cache miss rate is also high—for UMT this rate was small enough to be insignificant. MCB drives the IL1 rate up because of frequent branches in the inner loop over particles; every time the Monte Carlo scheme rolls the dice to find out what happens next, that's another branch. We have plotted L1D and IL1 miss rates with the same scale so that they can be directly compared. The L2 latency resulting from the combination of L1D and IL1 is likely the reason why MCB integer instruction rates can go as low as 0.4 per cycle per core.

The L1P (prefetch) buffer miss rate adds little new information. Only a fraction of loads that miss L1D or IL1 hit in L1P. Most go to L2. The L2 miss rate is significant but not extreme (it's close to the rate we saw for UMT). On BG/Q the L2 cache is the last stop before main memory. The memory bandwidth requirement for MCB comes out to about a quarter of the maximum. The bandwidth curves track the L2 miss rate curves fairly closely, much more than they did for UMT. This is because UMT allows unneeded threads to go idle for high thread counts, whereas MCB keeps its threads busy doing "unproductive" synchronization work. MCB is not bandwidth-limited on BG/Q, though it could cross that line on plausible future machines.

The final plot, at lower left in Fig. 8, shows heap memory sizes used by these runs. These were obtained using mpitrace because the memP tool would not work for MCB. The plotted estimated heap use per node is actually the maximum heap use across all tasks, multiplied by the number of tasks per node. This was a good estimate because the mesh and the particles are both well balanced among the tasks in this simple test problem.

MPI parallelism in MCB is over spatial domains. As the number of MPI tasks in our experiment

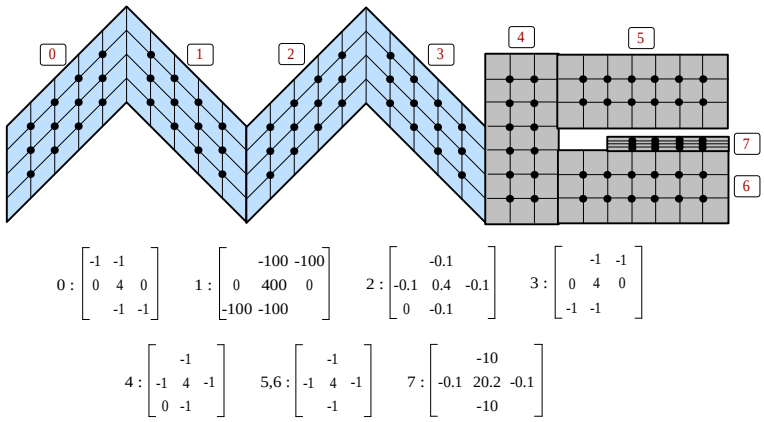

Figure 9: Grid layout and difference stencils for the AMG test problem. (This figure is copied from the AMG2013 source code distribution.[1])

decreases, the size of the mesh domains for each task increases by the same factor, so the total mesh size remaines the same. Each thread, however, carries its own copy of its mesh domain so the memory used for the mesh increases with the number of threads per task. In the limit of high thread count the memory allocated for the mesh can be much larger than the memory used for particles. We see this effect in the plot, where the curves that rise most steeply are the ones with a large number of zones per core.

AMG2013 Introduction and Hardware Counter Results

The AMG2013 proxy app uses algebraic multigrid to solve a linear system derived from difference stencils on the multiblock mesh shown in Fig. 9, with the added complication that different stencils are applied on different blocks of the mesh. It is based on code in the *hypre* linear solver library [7]. Algebraic multigrid itself is a complex algorithm which treats the mesh as unstructured, represents the matrix using a general sparse representation, and then constructs a series of coarsened representations of this matrix in a mathematically sophisticated way. This construction process is the "setup" phase. In the "solve" phase the multilevel representation is used to solve the linear system in $O(N)$ time, where N represents the total number of unknowns. Both phases are parallelized, both use a combination of MPI and OpenMP, but as with UMT and MCB the MPI implementation is more mature. Both forms of parallelism have been more completely optimized in the solve phase because this phase is more straightforward, because it operates on fixed data structures, because a single call to setup can be amortized over many calls to solve (if the user needs solutions for many different right hand side vectors), and perhaps because the solve has historically been seen as more mathematically interesting.

The solve is an iterative process. The benefit of using a multigrid method is that this stage can be completed in $O(1)$ iterations, independent of N, whereas most other iterative linear solvers require

increasing numbers of iterations for larger N. Setup is not iterative and merely sets the stage for the solve phase. Scaling the setup phase presents no mathematical difficulties, but building the complex, data-dependent structures is difficult to implement efficiently. Setup is harder to thread than the solve phase.

AMG is a more complex algorithm than either UMT or MCB, it can be more communication-bound, and our characterization results are less conclusive for it than for the other two proxy apps. We therefore used the UMT and MCB results to satisfy the milestone and omitted most of those for AMG. Conclusive or not, though, there are interesting trends here that suggest avenues for future study, and there are measurements that help corroborate some of our observations for UMT and MCB.

We do not have extensive results for AMG using the BG/Q hardware counters. Because scaling and global communication are such critical issues for AMG we put more effort into the mpiP results detailed in the sections below. We did run a few representative tests to measure the hardware counters to get an idea of their typical values but did not investigate their dependence on problem input parameters or machine configuration in any systematic way.

The values we will quote are for a run on 4 nodes of Vulcan with 64 MPI tasks using the (16×4) thread mode, and a similar but larger run on 16 nodes. In these runs AMG was even more integer-intensive than MCB. The setup phase used 99% integer instructions, the solve phase 92-93%. This is not surprising given the unstructured matrix representation with a single unknown per zone and the frequent transfers between different levels of refinement. The setup phase is responsible for building the solver infrastructure and includes control logic at the scale of individual matrix elements, whereas the solve phase runs with existing data structures and less fine-grained control logic, so it makes sense that solve is more floating-point intensive. Memory bandwidth was low during setup but was over 18 GB/s during solve, comparable to the highest rates seen with UMT. Though the solve phase as a whole is not bandwidth limited some portions of it may be, since the memory access intensity is presumably not uniform throughout the entire solve. Instruction issue rates ranged from 0.45 to 0.7 per cycle per core and were similar for setup and for solve phases of the same problem. L1D cache miss rates hovered near 10% and were likely a factor in limiting the instruction rate, but some of the tests suggest that L2 efficiency could be even more important. During the solve phase, loads that miss in L1D often hit in the prefetch buffer L1P, whereas during setup most L1D misses hit in L2. The derived statistics for cache behavior do not always vary in the expected directions, which suggests that some other limiting factor may also be in play. Instruction cache misses—which we did not measure—could be a factor during setup but are unlikely to amount to much during solve.

As well as telling us something about AMG, these results help confirm that similar numbers we saw for UMT and MCB were not unique to those particular applications. The high memory bandwidth of the solve phase was similar to rates we saw for UMT, and the high ratio of integer to floating point instructions was similar to that for MCB.

AMG2013 Weak Scaling and Threading

We structured the AMG weak scaling study a bit differently from the other two proxy apps. We did not vary the problem parameters, but we did test a large number of different thread modes for weak

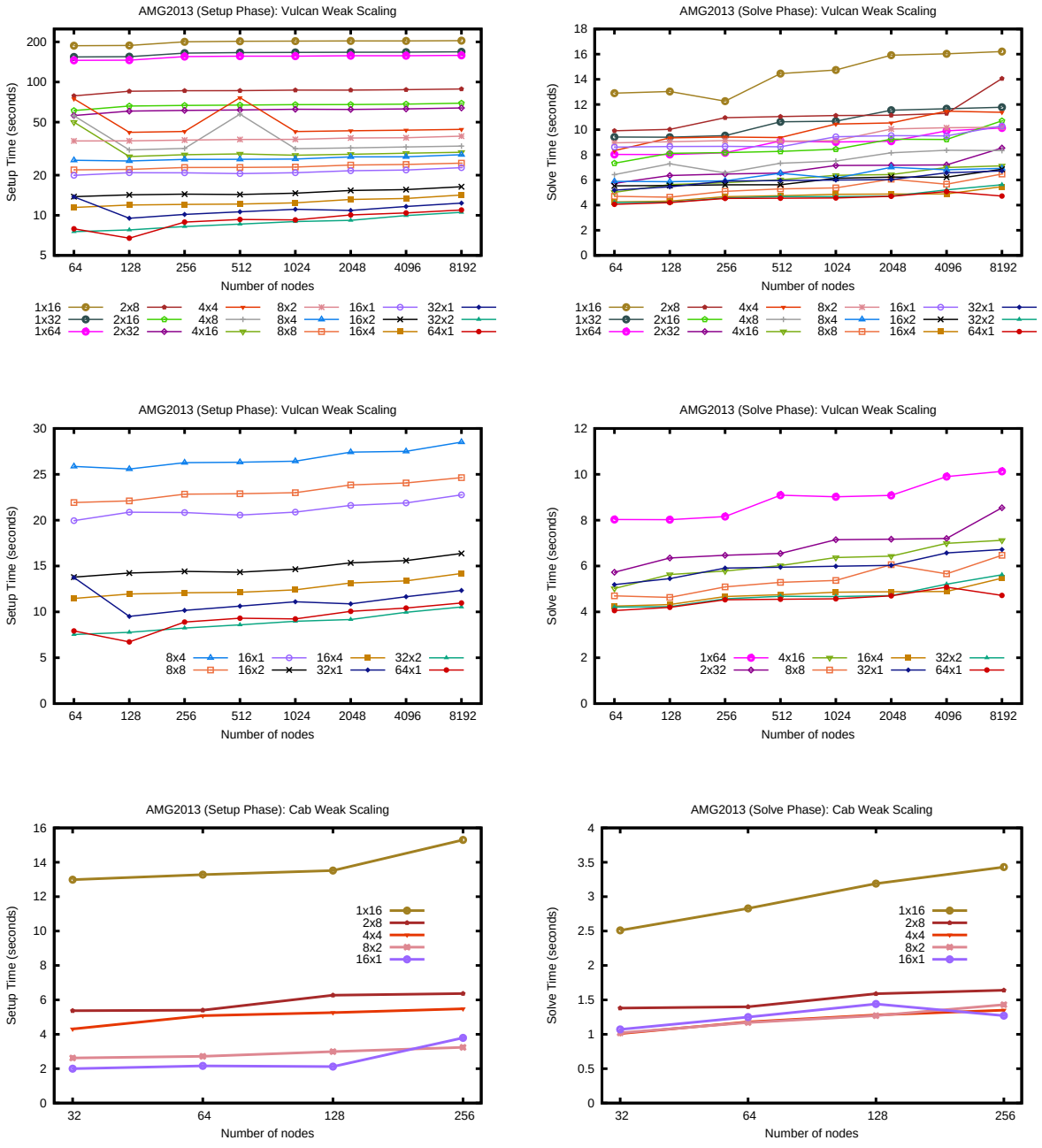

Figure 10: AMG weak scaling. The top row shows a large number of different thread modes on Vulcan, not all of which use the full 64 hardware threads on each node. Wall clock times rather than rates are shown so lower curves are faster. Some modes are extremely slow for the setup phase, so the top left plot uses a log scale. The middle row shows the same results, but limited to the fastest modes for each phase—these are not the same modes for setup and for solve. The bottom row shows similar results for Cab.

scaling on BG/Q. All tests were normalized to a $16 \times 16 \times 16$ domain on each physical core (multiply this by 16 to get the problem size per node). We kept the data per core constant but the data per MPI task changed depending on the number of OpenMP threads per task. For example, when using 2 MPI tasks per node and 8 OpenMP threads per task, the problem size per task becomes $32 \times 32 \times 32$. The problem size per core still remains the same at $16 \times 16 \times 16$.

Setup and solve were timed separately. The first row of Fig. 10 shows timings for many different thread modes, not all of which use the full 64 hardware threads on each node. A few individual runs seem anomalously slow which may be due to transient conditions or running on a slow node; we did not repeat these tests but we suspect the outlying results would change if we did. Ignoring the outliers, though, all thread modes show good MPI weak scaling, but the setup phase shows consistently poor performance for large numbers of threads. Note that we had to use a log scale for setup in order to show the slowest modes.

The second row shows only the fastest eight modes for each phase and does not need a log scale. Not all of the modes are the same for setup and for solve. Setup is fastest in modes with few OpenMP threads, while solve is fastest in modes that use all 64 hardware threads on each node. The conclusion is that not all parts of the setup phase have been efficiently threaded. The third row shows similar weak scaling timings for Cab. Fewer thread modes were tested but the results are similar to those for Vulcan and to the results for the other two proxy apps. Solve shows better thread performance than setup, but even for solve the (1×16) mode is anomalously slow.

AMG2013 Weak Scaling: Variability and mpiP Results

We know from the UMT study (Fig. 2) that run times on Cab are somewhat variable. With AMG we tested this variability in the weak scaling test series, focusing on the pure-MPI (16×1) mode. The first row of Fig. 11 shows the results for both the setup and the solve phase. The setup phase appears to be more variable than the solve, except for a single outlier in the solve tests at 256 nodes where one of the runs took more than twice as long as the others. We don't know the reason for this particular outlier—not even whether it was a delay on the network, or on one of the processors. Some variability in Cab timings for other performance tests has been traced to a few slow machine nodes. That explanation doesn't seem to work well here, though, since setup and solve phases of the same runs vary in different ways. Instrumentation of similar runs using mpiP, though, does provide at least a partial explanation for the general fact that some results are more variable than others.

During the setup phase there is a steady increase in the times spent both in communication and in computation, and communication is always a major portion of the time. This breakdown is shown in the middle row of Fig. 11. During the solve phase, though, communication times are a smaller but gradually increasing proportion of the total until 256 nodes are used, at which point the communication time appears to jump sharply. The breakdowns shown are averages over four runs, and much of this jump is due to one particularly slow run which inflated the average.

Whatever its cause, performance variability seems more strongly associated with communication-heavy phases of the algorithm, and manifests itself primarily through larger communication costs. In the final row of Fig. 11 we break communication times down by the type of MPI call and see that `MPI_Allreduce` has particularly bad scaling performance.

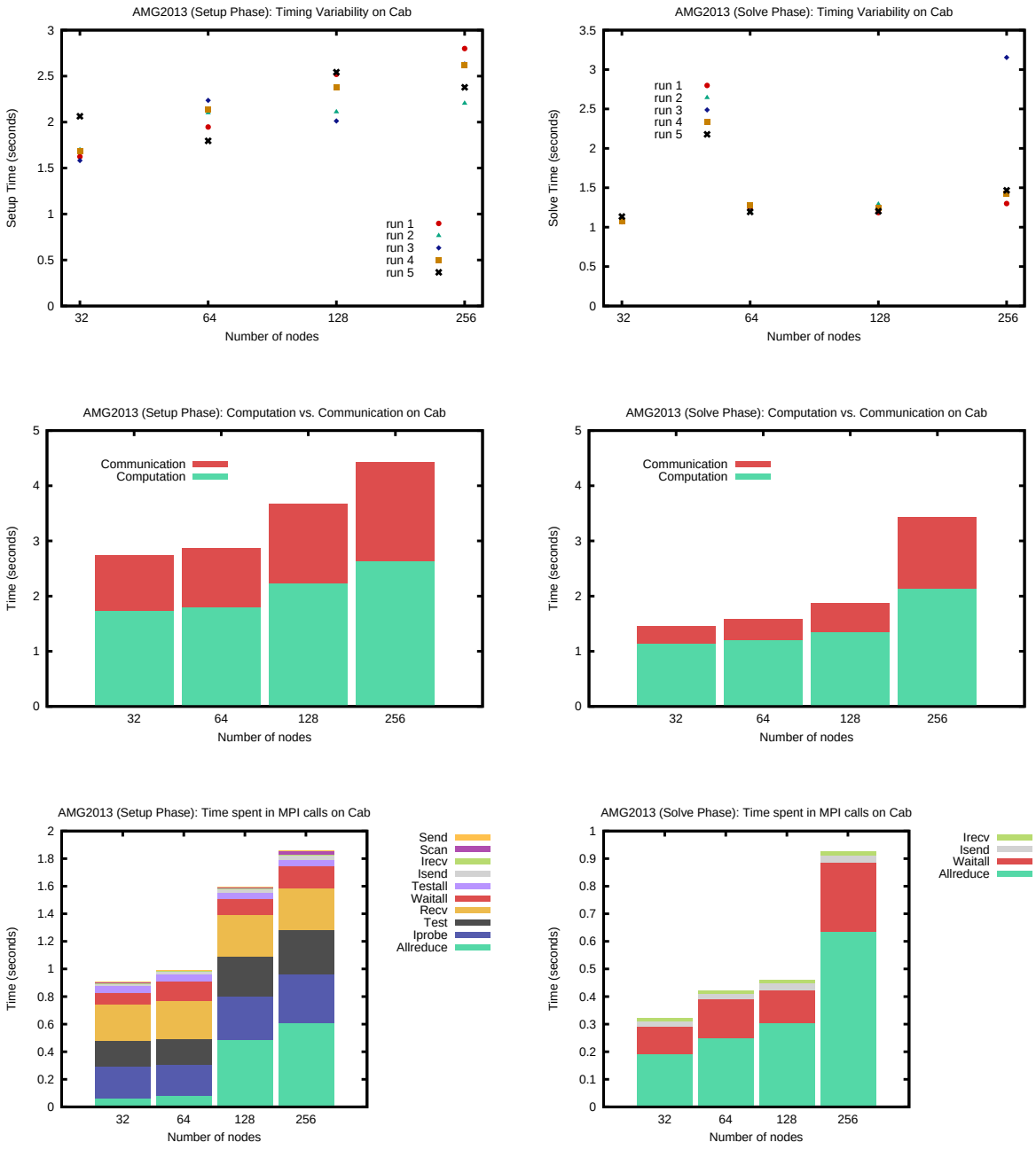

Figure 11: Top: Weak scaling timings on Cab in 16 x 1 mode showing multiple runs per test. Middle: Times spent in communication and computation per run (averaged over four runs). Bottom: Times spent in various MPI routines (aggregated over call sites for those routines).

In each case shown in Fig. 11 the cost attributed to computation tended to increase in tandem with communication but to a proportionally lesser extent. The same correlation appears in results for Vulcan presented below. It is not clear whether all of this correlated change is real—due perhaps to the increased number of multigrid levels the code builds for larger problem sizes—or whether some of it could be a measurement artifact. As evidence that it may be partly an artifact, note that the totals of communication and computation time as reported by mpiP don't match very closely with the total times shown on the row above and in Fig. 10. The two sets of timings are gathered by different mechanisms, with the mpiP results being aggregated and averaged over all MPI tasks. We would have to look more closely at how mpiP attributes costs to different phases if we wanted to better understand this apparent discrepancy.

We turn now to some results from Vulcan, where variability is not so much an issue but node counts can be much higher. There is an algorithmic option set in the AMG proxy app at build time (and in the *hypre* library from which it is derived) that we had to use in order to get good weak scaling performance at high processor counts. This option is `--with-no-global-partition`. At the time of this study this option was not the default in *hypre* because the *hypre* developers were concerned it might hurt performance for small numbers of processors. In our tests it produced such a strong improvement in setup times, though, that we recommend it for any *hypre* installation intended for large-scale computing. There was no observable effect on the time for the solve phase. The upper left plot in Fig. 12 shows timings on Vulcan using the pure-MPI (64×1) mode with and without this option. The upper right plot shows the breakdown by MPI calls with global partitioning (that is, with the `--with-no-global-partition` turned off). All of our other AMG timings in this report had this option turned on, including those in Figures 10 and 11 along with the second and third rows of Fig. 12.

The middle left plot of Fig. 12 shows the communication and computation costs on Vulcan (using `--with-no-global-partition`). The communication times are a smaller proportion of the total times than on Cab and are scaling fairly well. The middle right plot gives the corresponding breakdown by MPI call, and can be directly compared to the result with global partitioning above it. These show that the MPI implementations are very different for the two versions of the algorithm, and that heavy use of `MPI_Allgather` and `MPI_Allgatherv` contribute to the nonscalable behavior of the less efficient version. Note that these breakdowns combine the times for the setup and solve phases. Separate breakdowns for setup and solve for the more efficient version are in the bottom row, though the result for 2048 nodes is missing for solve because of a memory problem in mpiP itself. This last row of Fig. 12 can be directly compared to the similar results for Cab in the last row of Fig. 11.

AMG2013 Strong Scaling

We have only one strong scaling result for AMG in this study, which was run on Cab. This test started with a $32 \times 32 \times 32$ domain per physical core when running on 32 nodes and reduced down to $16 \times 16 \times 16$ domains on 256 nodes, maintaining a constant overall problem size. This is not a huge dynamic range but the results highlight many of the same effects we saw in the weak scaling tests.

Figure 13 is a log-log plot. With ideal scaling all of the curves would be straight lines with the same slope. The most obvious departure from this ideal, the increase in time for the solve phase at

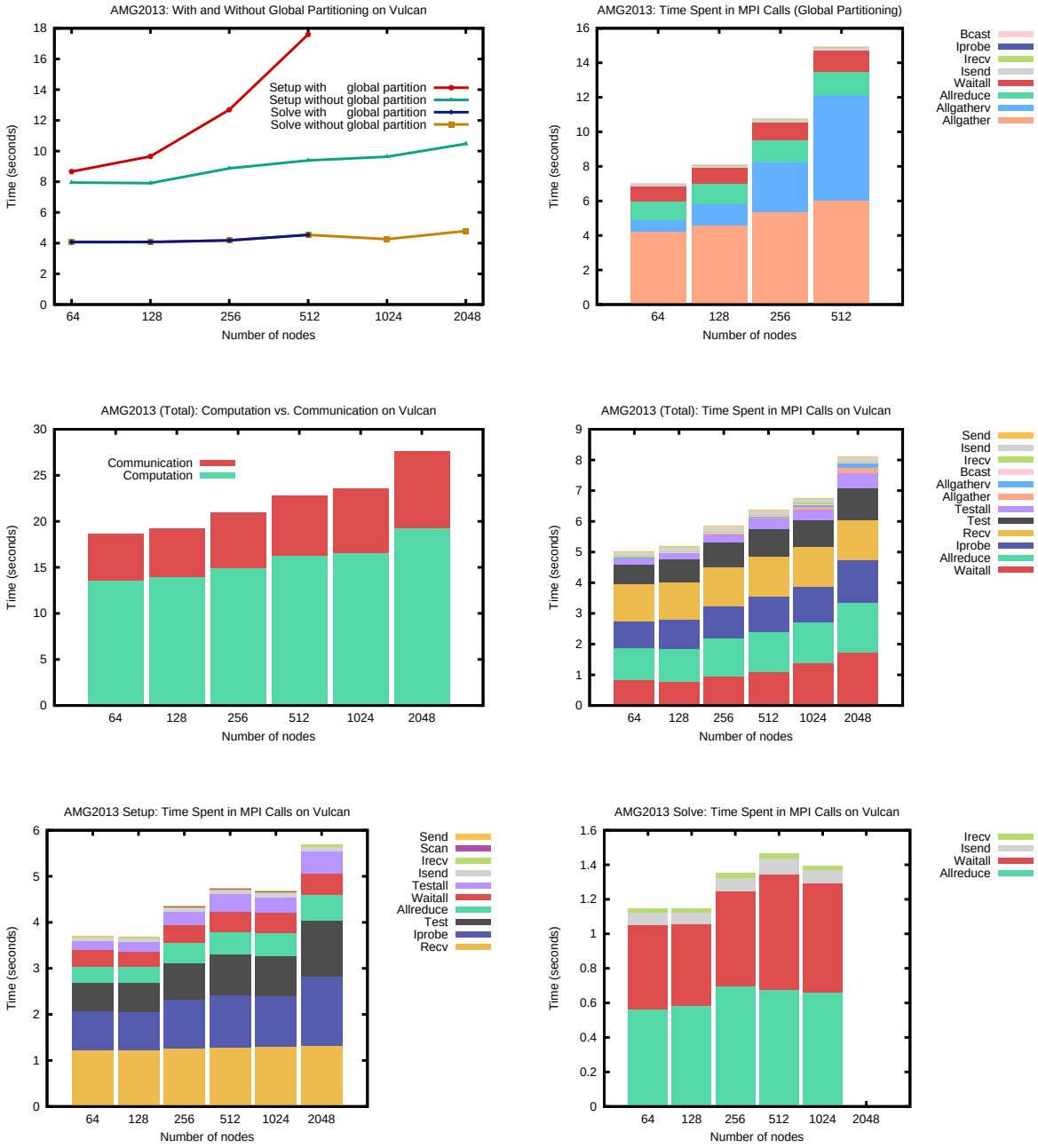

Figure 12: Top Left: Weak scaling runs on Vulcan in 64 x 1 mode with and without the –with-no-global-partition option. The setup without this option was not only slower, it ran out of memory for more than 512 nodes. Other plots show the communication and computation costs for the preferred version where this option is used, the breakdown by MPI routine for both versions, and the breakdown divided into setup and solve phases for the preferred version.

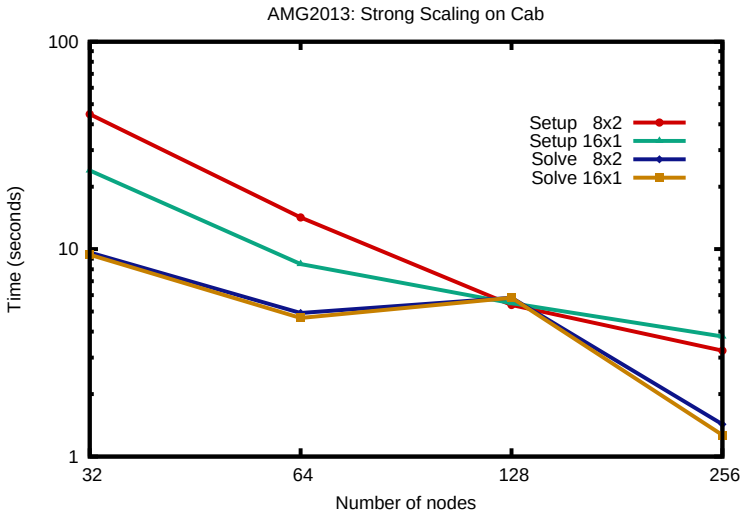

Figure 13: Strong scaling test of AMG on Cab showing setup and solve times and two different threading modes. Note that both axes are log scale.

128 nodes, may not be real—it may be an example of the variability of results we sometimes see on this machine. In any case we have no other explanation to offer for this anomaly. The solve times for 32, 64, and 256 nodes all fall on a nearly straight line. Not only that, they fall on the same straight line, due to the effective threading of the solve phase.

The curves for the setup phase are not as straight, nor are they as close to each other. At low node counts the (16×1) mode is considerably faster than (8×2) because the setup phase is not threaded as effectively. At higher node counts, though, this difference shrinks and then reverses itself. Both curves tilt upward slightly—meaning that they do not scale perfectly—but the (16×1) curve tilts up more. We believe this may be because setup includes more fixed costs and does not strong scale as well as solve. The (16×1) mode is running more MPI tasks and so shows this effect more clearly than (8×2).

BG/Q Torus Mapping Results for AMG

In the sections above we have presented information on MPI scalability, OpenMP threading, and on-node performance characteristics for all three of the proxy apps UMT2013, MCB, and AMG2013. We have used some standard tools, mainly mpiP and the hardware counter information available on BG/Q machines, but the focus has been on the apps rather than on the tools themselves.

In addition to those more standard results, though, we have also looked at some of these apps in more unusual situations and with more experimental performance tools. This shifts the focus away from evaluating the apps and toward evaluating these new techniques to see if they can tell us something interesting about how to understand performance on advanced machines. The remaining sections of this report will present a sampling of these experimental results and will be

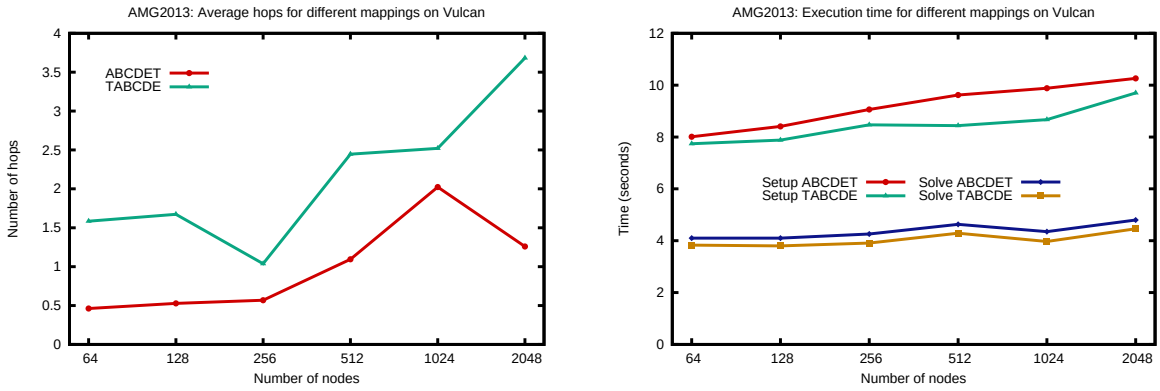

Figure 14: Changing to an alternative BG/Q torus mapping causes messages to take more hops through the network but decreases execution time, possibly by decreasing contention. These runs were done in the pure-MPI 64 x 1 mode.

less systematic than the sections above.

Figure 14 compares the performance of AMG2013 for two different schemes for arranging MPI tasks on the interconnect [8]. This is done by trying two different system-provided layouts on the 5D torus. Ideally we would expect that lowering the number of hops messages take through the network would reduce the amount of network congestion and hence improve performance. But here we see an inverse correlation. The default mapping ABCDET puts adjacent MPI tasks on the same node and gives many fewer hops than an alternative mapping TABCDE, but yields slower run times. Understanding contention on networks is challenging and at this point we do not have a good idea of why TABCDE does a better job with increased hop counts. One possible explanation is that the TABCDE mapping reduces the congestion on the injection FIFOs by reducing the number of source MPI processes trying to send messages to the same destination node. Another possible reason is that on Blue Gene/Q it is faster to move data off a remote process (presuming it is in cache) than from main memory to cache. So while ABCDET features lower hop counts, it also features more messages that are not off-node that involve moving data through memory.

Using NVRAM to Supplement Main Memory for UMT

The Catalyst machine at LLNL is similar to the Sandy Bridge TLCC2 machine Cab but uses 12-core Xeon E5-2695 v2 Ivy Bridge processors. Its most distinguishing features are large memory (128GB per node) and an additional 800GB of SSD NVRAM per node. Given that UMT with its six-dimensional discretization requires storage of large amounts of data, and that future architectures may provide less memory per processor and may require use of complex hierarchical memory systems, it was interesting to look at whether UMT could use the SSDs on Catalyst as a supplement to main memory. (Other Lab codes have begun to do so. [9])

The software package DI-MMAP, currently under development at LLNL [10], allows pages on the heap to migrate between NVRAM and main memory as needed. A buffer in main memory holds the currently resident pages. If this buffer is large enough to contain the entire heap, then the

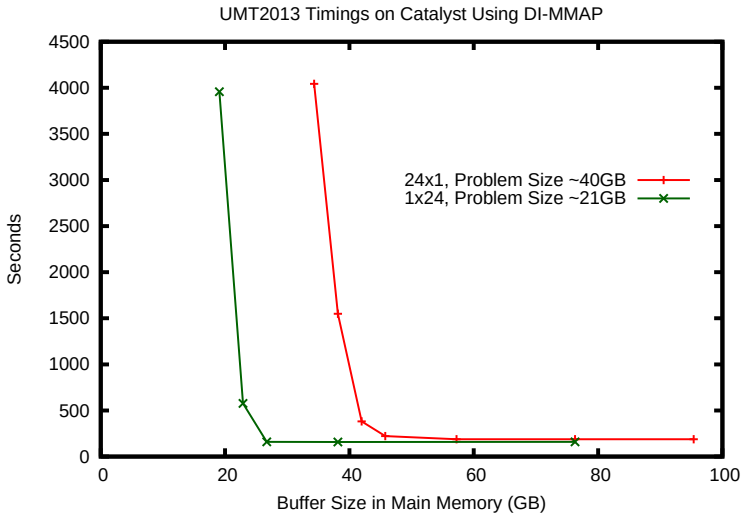

Figure 15: Performance on a single node of Catalyst as a function of the DI-MMAP buffer size in main memory. When the buffer size is too small, allocated pages migrate to NVRAM and execution time goes up. The large increases in run time observed for small buffers with both MPI and OpenMP parallelism show that the current implementation is not effectively managing memory for UMT.

application runs nearly as fast as it would normally (measurements with UMT2013 showed a slowdown of around 15% compared to the same tests run without DI-MMAP). If the heap is too large for the buffer then some pages will only exist in NVRAM at any given time and may be swapped into the buffer as needed.

The obvious approach was to allocate a buffer that nearly fills main memory on a node and then experiment with UMT2013 test problems larger than the buffer. This turned out to be more difficult than expected. UMT2013 is an idealized proxy app, and though it scales well to large numbers of MPI tasks it is not set up to handle unusually large memory sizes per task. The grid generator scales very poorly with increasing zone count per spatial domain. Though this initialization time is not counted as run time for the application itself, it quickly becomes prohibitive—hours to initialize a test that would take minutes to run. There are also built-in limits on the numbers of angles and energy groups. Rather than try to modify UMT to enable larger problem sizes, we decided to frame the test a different way. After all, dealing with a large main memory was not the point of the exercise.

Figure 15 shows timings for two series of tests, each one holding the problem size fixed while varying the size of the buffer allocated in main memory. This not only avoids the need to construct very large test problems, it also allows us to estimate the costs of swapping pages to NVRAM without the complication of changing other computational costs at the same time. Both test problems ran on a single node of 24 cores. One used 24 MPI tasks, the other used 24 OpenMP threads (though the code was built with support for MPI+OpenMP in both cases).

The run times approach a constant value for all buffer sizes large enough to contain the entire

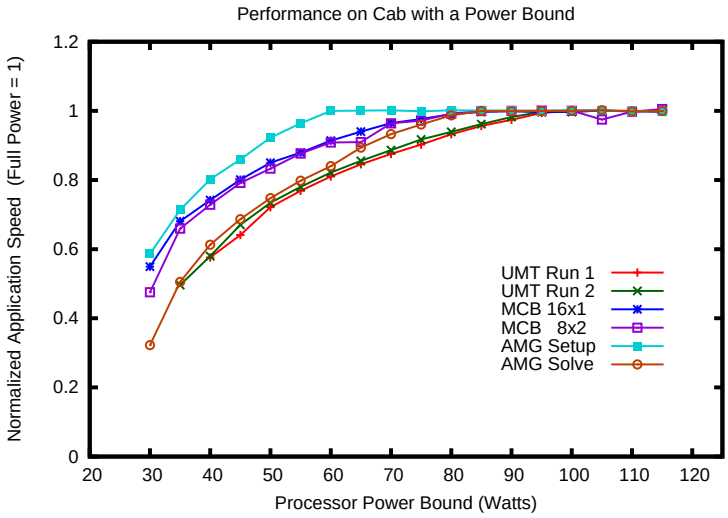

Figure 16: Performance of UMT2013, MCB, and AMG2013 on Cab as a function of a bound on processor power set using the MSR library. So that the codes can be compared to each other, for each curve the figure of merit for each run is divided by the figure of merit without any power limitation.

allocated heap. As the buffer becomes smaller than the heap size, though, run times increase dramatically. Even small amounts of memory moving to and from NVRAM can use more time than the rest of the algorithm combined.

This is a disappointing result and has not yet been explained. On the bright side is the fact that DI-MMAP worked at all for this code. This is its first use with MPI, and DI-MMAP developer Brian Van Essen had to make significant modifications in order to get it running. He believes there may be an interaction with MPI that causes pages to migrate more than necessary, so improvement in future versions of DI-MMAP is possible. On the other hand, the access patterns used by UMT2013, such as the fact that the code accesses almost all of its memory every timestep, may render this sort of automatic approach impractical. It may be necessary to modify the code in order to explicitly stage access to memory before it is needed, much as MPI codes can be structured to overlap communication and computation.

Power Bound Experiment

Available electric power is already a major design consideration for large parallel computers, and in the future it may be necessary to explicitly manage the power requirements of individual jobs. The reason is that power use varies according to the characteristics of particular codes and it is expensive to overprovision a computer center with power capacity that is not typically needed. It is not yet clear, though, what a power-aware scheduler or a power-aware application would need to look like in practice. We could imagine limiting the number of nodes available to power-hungry applications at any one time, or allocating more power to performance-critical sections of a single code.

The MSR library [11] on Cab allows a user job to set an upper bound on power available to each processor socket. As a simple experiment, we ran each of our three proxy apps over a range of power bounds in order to see how this limitation would affect performance. The results are shown in Fig. 16. Each of the three apps degraded gracefully as the bound was decreased, but they did so at different rates. To limit any variation due to running on different nodes of Cab, we ran all of the tests for each curve within a single batch script, so that all points on a curve represent results for the same set of physical processors. Different curves represent results from different batch jobs and so probably ran on different processors. Each curve is normalized so that speed without a power bound is 1.0.

The first two curves were for UMT and show that results from similar tests are repeatable. UMT makes heavy use of both floating-point and memory bandwidth and showed the strongest sensitivity to the power bound. Performance was constant for a bound of 95 watts or higher but declined steadily below that. The next two runs were for MCB using two different thread modes. The two modes gave similar performance, which was constant for bounds of 85 watts or higher. Performance at lower power levels does seem consistently slower for the (8×2) mode, but the differences are small enough that they could be due to running the two series of tests on different processors rather than to different power requirements of different thread modes. The two identical UMT curves differ by a comparable amount.

For AMG we plot the performance separately for the setup phase and the solve phase of each run. The solve phase does significant floating-point work and uses memory bandwidth as heavily as UMT. Its performance is constant at 85 watts and above, but below that degrades at a rate between those for UMT and MCB. The setup phase, though, does little floating-point work and has lower memory bandwidth requirements, and has the most forgiving power requirements. Setup ran at full speed all the way down to a 60-watt power bound.

To be clear, we did not place power bounds on DRAM itself, only on the CPU sockets. We also did not explore other possibilities: the MSR library allows setting separate power bounds for each socket, measuring the actual power usage directly, and adjusting power bounds on timescales of a few milliseconds. A complex heterogeneous application could control its power usage in complex, fine-scale ways. Whether it would be useful for it to do so is another question entirely, but it is at least a question worth considering.

Performance Tool Lessons Learned

The performance tools gave useful information but did not always work well—some of the problems were outright bugs. Some issues were described in more detail in earlier sections. Here is a quick summary of difficulties encountered:

- Building UMT with dynamically-linked libraries caused trouble for both mpiP and memP, though many features could be made to work.
- mpiP post-processing does not scale well without the `-l` option.
- memP can give incorrect heap sizes with run with threads or when run on Catalyst even without threads.

- memP shows the heap size used by the program, which may be significantly smaller than the total size of the memory pool allocated to the process by the operating system.
- memP never worked for MCB.
- At high instruction cache miss rates some derived quantities returned by the HPM library on BG/Q became unreliable with the default counter group 0. Using `HPM_GROUP=8` avoided this problem by explicitly including instruction cache misses.

Conclusions

This is a work in progress, and the conclusions we can draw from studying proxy apps alone can only be tentative. Further work will require comparison with production codes running more complex simulations with real-world applications. We must also understand how codes behave on additional architectures with accelerators and more complex memory hierarchies. Nevertheless a few interesting results are apparent from these proxy app experiments alone.

MPI scaling was generally good. This is not surprising. Codes and algorithms have been designed, studied, written, and rewritten for distributed-memory parallel computers for quite a few years now, not just in academic research but in the lab environment that produced these algorithms. The parallel machines themselves have also gone through several generations of evolution to better support programmatic applications. The significant differences between the machines examined here highlight the importance of consistency in node and interconnect performance for large-scale bulk-synchronous applications. A conclusion is that either that consistency must be maintained in future architectures, or the largest-scale applications will have to move away from bulk-synchronous programming models in order to adapt better to a changing runtime environment.

The results for on-node performance with OpenMP show that the current threading models in all three proxy apps are effective only for small numbers of threads. UMT2013 may make more efficient use of memory bandwidth than earlier UMT versions, but it is still near the bandwidth limit for BG/Q. The solve phase of AMG2013 also approached the memory bandwidth limit. Cache performance was reasonable for all three apps but differences in cache miss rates may explain some of the performance differences we observed. Both threading and memory management are likely to drive future developments in code and compiler design; these issues appear to be less settled than MPI parallelism.

The high ratio of int to float instructions shows how integer instructions cannot be considered an afterthought for complex codes, even for a mesh-based code like UMT2013. Integer instruction rate was the most important factor limiting UMT performance, though floating-point operations were also significant. The other two proxy apps were even more lopsided, to the point where floating-point operations seemed almost irrelevant. Future architecture choices should take this into account, and not focus too much on features such as floating point vectorization that may only benefit a limited set of codes.

On the issue of how to use NVRAM and other new additions to the memory hierarchy, we have only scratched the surface. Preliminary experiments shown here were not promising, and suggest that major shifts in algorithm design may be needed. The experiment involving BG/Q torus

mappings showed that complex hardware environments can behave in counterintuitive ways, and that the most obvious task layouts may not be the optimal ones. The power bound experment shows that different codes behave in different ways under limited power. If power restrictions become more important on future machines it may be necessary to take such variations into account.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Helpful advice was given by Todd Gamblin and Ian Karlin on interpreting performance data, by Chris Chambreau on the use of mpiP and memP, by Paul Nowak on UMT, Nick Gentile on MCB, and Ulrike Yang on AMG. Brian Van Essen provided access and assitance with DI-MMAP and Barry Rountree with the MSR library. Rob Neely, Tom Brunner, Bert Still, Mike Zika, and Lori Diachin made useful comments.

References

[1] `https://asc.llnl.gov/CORAL-benchmarks/`.

[2] Walkup, R. E., "MPI Wrappers for BGQ," IBM Advanced Computing Technology Center, Yorktown Heights, NY, unpublished document available on BG/Q systems (2013).

[3] Vetter, J., and Chambreau, C., "mpiP: Lightweight, Scalable MPI Profiling," `mpip.sourceforge.net` (2014).

[4] Chambreau, C., "memP: Parallel Heap Profiling," `memp.sourceforge.net` (2010).

[5] Nowak, P. F., and Nemanic, M. K., "Radiation Transport Calculations on Unstructured Grids Using a Spatially Decomposed and Threaded Algorithm," in *Proceedings of the International Conference on Mathematics and Computation, Reactor Physics and Environmental Analysis in Nuclear Applications*, Madrid, Spain, September 27–30, 1999, Vol. 1, p. 379.

[6] Nowak, P., "Deterministic Methods for Radiation Transport: Lessons Learned and Future Directions," *ASC Workshop on Methods for Computational Physics and Modern Software Practices*, March 2004, Lawrence Livermore National Laboratory report UCRL-CONF-202912.

[7] `https://computation.llnl.gov/project/linear_solvers/`.

[8] Bhatele, A., Gamblin, T., Langer, S. H., Bremer, P.-T., Draeger, E. W., Hamann, B., Isaacs, K. E., Landge, A. G., Levine, J. A., Pascucci, V., Schulz, M., and Still, C. H., "Mapping Applications with Collectives Over Sub-Communicators on Torus Networks," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '12,* IEEE Computer Society (2012). Lawrence Livermore National Laboratory report LLNL-CONF-556491.

[9] Van Essen, B., Pearce, R., Ames, S., and Gokhale, M., "On the Role of NVRAM in Data Intensive HPC Architectures: an Evaluation," in *IEEE International Parallel & Distributed Processing Symposium* (2012).

[10] Van Essen, B., Hsieh, H., Ames, S., Pearce R., and Gokhale, M., "DI-MMAP–a Scalable Memory-map Runtime for Out-of-core Data-intensive Applications," *Cluster Computing* (2013).

[11] Rountree, B., Ahn, D. H., de Supinski, B. R., Lowenthal, D. K., and Schulz, M., "Beyond DVFS: A First Look at Performance Under a Hardware-Enforced Power Bound", in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (2012).